Construction of a Peephole Optimiser

David Alex Lamb

August, 1980

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-80-141

Construction of a Peephole Optimiser

Accession For	
NTIS CRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

David Alex Lamb

August, 1980

Computer Science Department
Carnegie-Mellon University

Abstract

The design and construction of a peephole optimiser is described. The optimiser consists of a database of patterns, a translator which translates the patterns into an implementation language, and a pattern-matcher skeleton into which the generated pattern code is inserted. ~~We~~ argue that this three-part split is a good way to approach the construction of certain kinds of programs.

[illegible]

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

Table of Contents

List of Figures

1 Introduction and Background

This paper describes the design and implementation of a pattern-driven peephole optimiser. The optimiser was done as part of the Ada Charrette compiler project at Carnegie-Mellon University [5]. The Charrette compiler translates a subset of Preliminary Ada [4] into VAX 11/780 assembly language [3]. The main purpose of the compiler project was to explore potential implementation difficulties of the Preliminary Ada language. Consequently, little attention was paid to generating efficient local code. The code generator treats the VAX as a simple stack machine. In most cases it uses a single code template for each language construct, ignoring contextual information that could result in choosing a more efficient code sequence.

About two-thirds of the way through the project we discovered that the code generated by the compiler for even moderately large examples was sufficiently verbose that it was difficult for us to deduce what the code was doing; this made it rather difficult to debug the compiler. We decided at this point to implement some form of peephole optimiser, principally to reduce the size of the generated code.

The subject of peephole optimisation has been dealt with more formally and more completely elsewhere [8, 2]. The interesting aspect of this paper is not the treatment of peephole optimisation *per se*, but the method used to construct the optimiser.

Section 2 describes the compiler and shows where the optimiser fit into the rest of the project. Section 3 describes the structure of the optimiser. Section 4 evaluates the optimiser and the approach used to build it.

This paper contains a number of examples using VAX assembly language. Familiarity with some typical assembly language would be useful; in most cases the VAX-specific details are not important.

2 The Compiler

The compiler consists of several phases, each of which performs some kind of transformation of the program from one intermediate form to another. For the purposes of this paper, the relevant compiler phases are the code generator and the output phase. The code generator takes as input a tree-based representation of the program and produces as output a doubly-linked list of nodes representing individual assembly-language instructions [6]. The output phase transforms this list into VAX-11/780 assembly language.

The peephole optimiser is placed between the code generator and the output phase. It transforms the program emitted by the code generator into an equivalent but shorter program. The intermediate language used to represent the input to the optimiser is identical to the output language; this allows us to omit the optimiser. In fact, the Charrette project went for several months (most of its lifetime) without the optimiser.

2.1 Intermediate Languages

Within the running compiler, each of the internal forms of a program is some kind of tree or graph. Nodes within the graph are typed. For example, in one form there are **tree** nodes representing the parse tree of the program and several kinds of **symbol** nodes representing symbol table information. Each node contains a number of *attribute/value pairs*.

In order for a human to examine these internal structures they must have a textual representation. The notation we used is called LGN, for *Linear Graph Notation* [1]. Each node has a label which is denoted by an identifier followed by a colon. Each node also has a type which determines the set of attributes it may have. The general shape of the LG representation of a node is shown in Figure 1. In this example, we have a node representing a symbol table entry for a variable in some program. The node has a label, VAR1, so that other nodes may point to it. Its node type is VARIABLE_NODE, indicating that it is intended to represent some kind of program variable. It has a TYPE attribute which points to the symbol table entry for its type, and an ALLOCATION attribute which indicates how storage for the variable is to be allocated at runtime.

```
var1:   variable_node
        (type       type1:)     ! pointer to type descriptor
        (allocation static)     ! could be dynamic or controlled
```

Figure 1: Characterisation of LGN

2.2 Machine Code Representation

The input to the optimiser phase is a doubly-linked list of OBJECT and LABEL nodes representing machine instructions and labels; we refer to this intermediate form as the CODE representation. An OBJECT node represents a machine instruction or an assembler pseudo-operation. It has an OPCODE attribute which contains the assembler mnemonic for the instruction, and an OPERAND attribute which is an array of pointers to ADDRESS or LABEL nodes representing its arguments. A LABEL node represents an assembly language label; it has a NAME node giving the text to be used for references to the label. OBJECT and LABEL nodes have PREV and NEXT attributes which are used to link them together in the order the instructions should appear in the assembly language program. An ADDRESS node has a MODE attribute indicating the addressing mode it represents; its OPERAND attribute is an array of integers representing various portions of the address.

An example of machine instructions and the corresponding CODE form is shown in Figure 2. L1 is a LABEL node used as the target of branches to node L2. L2 is an OBJECT node representing the ADDL2 instruction; L3 and L4 are its arguments. L5 is another OBJECT node representing a branch instruction; its argument is a pointer to the LABEL node L1. L1, L2, and L5 are linked together via their PREV and NEXT fields.

```
        Loop:   ADDL2   (SP)+,4(R0)
                BEQL    Loop

L1:     LABEL (NAME "Loop") (PREV L0:) (NEXT L2:)
L2:     OBJECT (OPCODE ADDL2)(OPERAND L3: L4:)(PREV L0:) (NEXT L5:)
            ADDRESS (MODE AutoIncrement)(OPERAND SP)
L4:         ADDRESS (MODE Displacement) (OPERAND R0 4)
L5:     OBJECT (OPCODE BEQL) (OPERAND L1:) (PREV L2: NEXT L6:)
```

Figure 2: CODE form of machine instructions

3 The Optimiser

The peephole optimiser examines groups of adjacent instructions and tries to replace some sequences with better ones. There are two pieces to the optimiser. The first part is a collection of pattern matching and replacement routines which implement the optimisation transformations. The transformations are described in a notation resembling VAX assembly language. A translator compiles these patterns into code in the implementation language, BLISS-10 [7]. The second part of the optimiser is a code skeleton provides control for the application of patterns, and a collection of subroutines used in the pattern match process.

The remaininder of this section describe the pattern language, the optimiser skeleton, the translator which compiles the patterns, and the particular set of patterns used in the Charrette optimiser.

3.1 The Pattern Language

A pattern consists of a *match* portion and a *replacement* portion. Both portions are written in a pattern language based on VAX 11/780 assembly language. Symbols of the form &N, for some integer N, are pattern variables. In the match portion of the pattern they will match anything that can be placed in the corresponding position of the VAX instruction. They are used in the replacement portion of the pattern to fill in parts of the replacement instructions which are to be copied from the original instructions.

An example of such a pattern is shown in Figure 3. The first two lines after the **BEGIN** (containing a PUSHAL and a SUBL2) are a pattern which will match a pair of adjacent VAX assembly language instructions. The first effectively pushes the contents of a register (&2) onto the stack. The second instruction subtracts a literal (&3) from the data word at the top of the stack; the fact that this is a literal is indicated by the preceding "$". This sequence can be replaced by a new instruction where the displacement field of the argument to the PUSHAL is changed to contain the negative of the old literal.

```
BEGIN   # Local variable accessing
        pushal  0(&2)
        subl2   $&3,(sp)
        -->
        pushal  &negate(&3)(&2)
END
```

Figure 3: Example peephole optimisation pattern

The pattern language allows arbitrary code from the implementation language to be inserted in either the match or replacement portion of the pattern. These *escapes* typically take the form of calls on subroutines; whenever a new escape is created, the definition of the subroutine must be added to the code skeleton described in Section 3.2. In the match portion, such an escape is used to indicate some match condition that would be difficult to express in terms of assembly-language syntax. In the replacement portion, such escapes are used to indicate the construction of something that would have been difficult to build out of pieces of the instructions that appeared in the match portion.

The &Negate in Figure 3 is an example of a replacement escape. It causes the negative of the value of its argument, rather than the argument itself, to be placed in the output pattern. Figure 4 shows the application of this pattern.

```
Before                          After

pushal  0(R9)                   pushal    -10(R9)
subl2   $10,(SP)
```

Figure 4: Use of Pattern From Figure 3

Figure 5 shows the use of escapes in the match portion of a pattern. The text after the vertical bar in the first line of the pattern is an implementation-language expression which is expected to return a value of **true** or **false**. If a match happens based on the portion of the pattern built from the text before the bar, then any tests specified by the escapes are performed. The pattern matches only if these additional tests also succeed. Thus the pattern in the example matches any JLBS ("jump if lower bit set") instruction whose first argument is an odd literal; i.e. one whose lower bit is set. It replaces this conditional jump by an unconditional one.

```
BEGIN   # Jump on TRUE
        jlbs    $&1,&2: | IsOdd(&1)
        -->
        jbr     &2
END
```

Figure 5: Escapes in patterns

3.2 The Optimiser Skeleton

This section describes the compiled form of the patterns, and the pattern matching algorithm.

A pattern is represented by a record structure similar in form to the declaration shown in Figure 6. The name of the pattern is included for debugging purposes; it is created by the pattern translator. The match and replacement portions of the pattern are transformed into arrays of pointers to subroutines. All of the match-portion escape expressions are collected into a boolean subroutine whose address is also placed in the pattern record structure.

```
type Pattern is
    record
        Name: string;
        NumberOfMatches,NumberOfReplacements: constant integer;
        Matches: array(1..NumberOfMatches) of access procedure;
        Replacements: array(1..NumberOfReplacements) of access procedure;
        Escapes: access procedure;
    end record;
```

Figure 6: Pattern Record Structure

There is one entry in the array Matches for each instruction in the match portion of the pattern. Each such subroutine consists of inline code to recognise the particular opcode used in the pattern, followed by calls on subroutines to try to match each of the operands of the instruction. Similarly, there is one entry in the array Replacement for each instruction in the replacement portion of the pattern. Each such subroutine consists of inline code to build an OBJECT node representing the replacement instruction, followed by calls on subroutines to fill in each of the operands of the instruction with ADDRESS nodes.

The pattern application routine maintains a cursor pointing into the doubly-linked list of instructions. It attempts to match each possible pattern at the current cursor position. If all matches fail it advances the cursor to the next instruction in the list and tries again. If a pattern matches, it unlinks all the instructions matched, builds a list of replacement instructions, links the replacement into the main list, and resumes pattern matching with the first instruction of the replacement list. In both cases matching starts again with the first pattern.

Matching of an individual pattern proceeds backwards rather than forwards. Given a particular cursor position and pattern, the matcher compares the instruction at the current cursor position with the last instruction of the match portion of the pattern. If this match succeeds, the matcher backs up in both the pattern and the instruction stream, until either the pattern is exhausted or a mismatch occurs. The reason for matching backwards is that optimisation replacements may allow further patterns to match earlier in the instruction stream. Without this backward match, the matcher must make several passes through the instruction stream in order to find all possible matches.

3.3 The Pattern Translator

The translator is a SNOBOL4 program that compiles the patterns into subroutines and data declarations in the implementation language. It makes some effort to share code among different patterns. If two or more different patterns contain instances of the same instruction, only one subroutine is generated to represent both instructions. If two different instructions have instances of the same operand, only one subroutine is generated for both instances. "Same" means that the two items have identical text, ignoring spacing, comments, and case of letters. The text from the pattern is used as an index into a SNOBOL table whose elements are the names of routines that implement the corresponding portions of the pattern.

3.4 The Patterns

There are an extensive set of patterns whose main purpose is to eliminate unnecessary PUSH instructions. Since the code generator treat the VAX as a pure, zero-address stack machine, code sequences of the form shown in the left column of Figure 7 are very common. The "`(sp)+`" indicates that the first operand of the op instruction is found by popping the top element from the stack. This sequence can usually be replaced by the one shown in the right column.

```
Before                        After

push   arg1                   push   arg1
push   arg2                   op     arg2,(sp)
op     (sp)+,(sp)
```

Figure 7: Typical PUSH optimisation

This kind of optimisation is even more important than indicated by this example. Ada is a block-structured language requiring some form of display. The typical code to access a local variable is shown in the "before" portion of Figure 8. The optimiser allocates two registers to cache the top two levels of the display; this allows the sequence to collapse to the "after" portion of Figure 8. This transformation is encoded in four patterns. Two of them set up the cache registers, and two others look for uses of the first two levels of the display.

```
Before
        push    Level(DisplayPointer)   ; pointer to activation record
        push    $Offset                 ; constant offset within AR
        addl2   (sp)+,(sp)              ; form address
        pushl   *(sp)+                  ; get contents of variable

After
        push    Offset(CacheRegister)
```

Figure 8: Local variable accessing

There are collections of patterns to take advantage of unusual VAX instructions. For instance, on entry to each new scope the compiler generates code to allocate and zero out new local storage. The zeroing is done with a complicated multiple-operand instruction called MOVC5, which is normally used to move and pad character strings. In many cases, if the size of storage is a small integer, one or two CLEAR instructions can accomplish the same task with less code space.

Much of the power of the optimiser is obtained from the interaction of several patterns. The PUSH patterns eliminate most of the unnecessary stack manipulations. Other patterns are written assuming that any possible PUSH optimisations have already been done.

4 Discussion

4.1 Relation to Other Work

The CODE internal form is based on the data structures used in the FINAL phase of the BLISS-11 compiler [8]. The Linear Graph Notation and the tools used to manipulate it were taken from the Production Quality Compiler Compiler (PQCC) project at Carnegie-Mellon university.

Others have built much more sophisticated peephole optimisers. Davidson and Fraser describe an optimiser based on patterns derived from a formal machine description [2]. The BLISS-11 compiler does crossjumping and branch/jump resolution, in addition to performing the sort of adjacent-instruction comparisons described here. In our case branch/jump resolution was done for us by the VAX UNIX assembler, but crossjumping was not done.

4.2 Ease of Use

The particular style of pattern we used was very easy to understand. It was easy to notice places in the assembly code where poor code sequences were being generated, and to devise new patterns to improve these sequences. This happened because the notation used to describe the pattern transformations was almost the same as that of the assembly language programs being transformed.

On the other hand, it was often necessary to write several patterns for one conceptually simple situation. For example, one simple optimisation is to change a MOVE of a constant zero into a CLEAR. The VAX has several storage units (bytes, words, longwords), each with its own collection of instructions. Our approach required a separate pattern for each storage size. This kind of situation is one where some more formal notion of the action performed by each instruction would have helped.

The developer of the patterns was responsible for ensuring that a pattern would match only when appropriate. For example, a pattern that removes a label must ensure that no instructions point to the label, by calling pattern-escape routines. The developer was also responsible for sorting the patterns in the "right" order, and for trying to avoid interference between patterns. This was usually not a problem, but a small number of patterns were special cases that prevented more general patterns from matching. A multiple-instruction pattern might expect to find a MOVE of a constant in the middle of the pattern. If some particular instance has a constant of zero, and MOVE 0 is changed to CLEAR, then the larger pattern will not match.

4.3 Manpower Costs

The initial version of the generator, optimiser skeleton, and patterns took one person a total of five normal working days. This intial version did not handle labels and had almost none of the auxiliary functions; the pattern match algorithm was a straightforward front-to-back match that required several passes for convergence. Subsequent revisions to bring the system to its present state took another two to three weeks, for a total of about twenty man-days.

The revisions included one complete rewrite of the pattern-application routines, to convert from forward matching to backward matching. This rewrite took only two normal working days, about evenly split between the generator and the pattern matcher. One or two of the patterns had to be re-ordered after this change, but no change to the patterns themselves was needed.

4.4 Numbers

The current version of the optimiser reduces the size of the output from the code generator by 40 to 50 percent. The resulting code comes reasonably close to what one would expect from a stack model of the target machine and is fairly easy to understand.

It is difficult to measure the speed of the optimiser; the execution time is dominated by the time needed to read in the original program and write out the optimised program. The I/O time for a twelve-page Ada desk calculator program was about one minute; the time between the end of the reading of the input and the start of writing the output was about two seconds of CPU time for this test program.

Module	Size (words)			Percent of Total		
	code	data	total	code	data	total
pattern matcher	9410	564	9974	18	8	17
patterns and escapes	13267		13267	26		23
LGN manipulation	12879	3234	16113	25	46	28
I/O routines	2613	334	2947	5	5	5
debugger	12984	2840	15824	26	41	27
	-----	-----	-----			
total	51153	6972	58125			

Figure 9: Size of the Optimiser

The storage space required by the optimiser is shown in Figure 9. The term "words" refers to 36-bit PDP-10 words. The numbers for the data include only fixed data; several tens of kilowords would be needed to hold the internal representation of a medium-sized test program.

The final version of the optimiser had 53 patterns, built out of 237 subroutines. The breakdown of the pattern subroutines was 69 for matching instructions or labels, 35 for matching operands, 33 for pattern escape subroutines, 65 for constructing new instructions, and 35 for constructing new operands.

5 Conclusion

This paper has described the design and construction of a peephole optimiser used in an experimental compiler. The optimiser was needed as a tool for developing the rest of the compiler, rather than as a research component of the project.

The optimiser was developed in three pieces: a notation for expressing the optimisations to be performed (the pattern language), a translator that mapped the pattern language into an implementation language (the generator), and an environment in which to run the patterns (the skeleton).

The closeness of the pattern language to assembly language allowed the designers to quickly and easily add new optimisations by observing compiler output, without regard to the details of how the pattern matching worked. The generator translated these patterns into a form that could be executed fairly efficiently. The three part split of patterns, generator, and skeleton, and the well-defined interface between these parts, allowed the project to be completed at a small manpower cost, and resulted in a flexible and easy-to-use program.

References

[1] B. M. Brosgol, J.M. Newcomer, D.A. Lamb, D. Levine, M. S. Van Deusen, and W.A. Wulf.
$TCOL_{Ada}$: Revised Report on An Intermediate Representation for the Preliminary Ada Language.
Technical Report CMU-CS-80-105, Carnegie-Mellon University, Computer Science Department, February, 1980.

[2] J. W. Davidson and C. W. Fraser.
The Design and Application of a Retargetable Peephole Optimiser.
ACM Transactions on Programming Languages and Systems 2(2):191-202, April, 1980.

[3] Digital Equipment Corporation.
VAX-11/780 Architecture Handbook.
1977.

[4] J.D. Ichbiah, J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Brueckner, B.A. Wichmann.
Reference Manual for the Ada Programming Language.
SIGPLAN Notices 14(6):1, June, 1979.

[5] J. Rosenberg, D. A. Lamb, A. Hisgen, and M. S. Sherman.
The Charrette Ada Compiler.
In *Symposium on the Ada Programming Language.* ACM, Boston, 1980.

[6] M.S. Sherman, A. Hisgen, D. A. Lamb, and J. Rosenberg.
An Ada Code Generator for VAX 11/780 with Unix.
In *Symposium on the Ada Programming Language.* ACM, Boston, 1980.

[7] W.A. Wulf, D.B. Russell, and A.N. Habermann.
BLISS: a Language for Systems Programming.
Communications of the ACM 14(12):780-790, December, 1971.

[8] W. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs, and C.M. Geschke.
The Design of an Optimizing Compiler.
American-Elsevier, 1975.